Mixed 0-1 Programming by Lift-and-Project in a Branch-and-Cut Framework

Egon Balas [1]

Sebastián Ceria [2]

Carnegie Mellon University

PITTSBURGH, PENNSYLVANIA 15213

Graduate School of Industrial Administration

WILLIAM LARIMER MELLON, FOUNDER

Management Science Research Report #MSRR-603

Mixed 0–1 Programming by Lift-and-Project in a Branch-and-Cut Framework

Egon Balas [1]

Sebastián Ceria [2]

Gérard Cornuéjols [1]

February 25, 1994

Accesion For	
NTIS CRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

[1]Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA 15213.

[2]Graduate School of Business, Columbia University, New York, NY 10027. The research was performed while this author was affiliated with Carnegie Mellon University and with CORE, Université Catholique de Louvain, Louvain-la-Neuve, Belgium.

This work was supported by NSF grant DDM-9201340, ONR grant N00014-89-J-1063 and a grant from Bellcore INI to Carnegie Mellon University.

Abstract

We investigate the computational issues that need to be addressed when incorporating general cutting planes for mixed 0–1 programs into a branch-and-cut framework. The cuts we use are of the lift-and-project variety. Some of the issues addressed have a theoretical answer but others are of an experimental nature and are settled by comparing alternatives on a set of test problems. The resulting code is a robust solver for mixed 0–1 programs. We compare it with several existing codes.

1 Introduction

For large scale integer programs, a straightforward use of branch-and-bound is often computationally prohibitive. In 1983, Crowder, Johnson and Padberg [8] demonstrated that such large scale problems can sometimes be solved to optimality in reasonable time by strengthening the integer programming formulation with automatic problem preprocessing and cut generation before applying branch-and-bound. Their work involved pure $0-1$ programs. In 1987, Van Roy and Wolsey [15] carried out a similar scheme for mixed $0-1$ programs. The successes reported in these two papers had a considerable impact on recent practice in integer programming. In both cases, however, each cut only strengthens one (or a few) constraint(s) while ignoring the rest of the formulation. As a result, the combinatorial structure of the underlying problem may not be exploited. To remedy this limitation, the popular line of research in recent years has been to consider classes of problems with special structure and to devise combinatorial cut generation procedures for each case. This approach has led to a number of impressive successes. However, its limitation is that each class of combinatorial problems requires a special purpose algorithm. A different approach, which was originated by Gomory [9] is an automatic generation of cuts based on the full formulation. With this scheme, the structure of the integer program may be exploited even when this structure is not apparent to the user. The Gomory cuts fell out of favor due to poor computational experience reported in the sixties, but nowadays this disenchantment is probably excessive in view of the greatly improved linear programming codes that are available. Recently, lift-and-project cuts [3] have been shown to be an effective way of strengthening mixed $0-1$ programs: not only do they dominate the classical mixed integer cuts of Gomory, but they are also numerically more stable. We will use them extensively in the work presented in this paper.

Another significant recent development in computational integer programming is the branch-and-cut approach introduced by Padberg and Rinaldi [14]. In branch-and-cut, the automatic cut generation is performed not only prior to starting branch-and-bound but also at each node of the enumeration tree. For this approach to be successful, it is important that the cuts generated at a node of the enumeration tree be valid at all the nodes. So far, this requirement has limited the types of cuts used in branch-and-cut to combinatorial cuts. In this paper, we show how to use lift-and-project cuts within a branch-and-cut algorithm. This combination of general cutting planes with branch-and-cut results in a versatile procedure for solving mixed integer programs. Our procedure does not require information about problem-specific structure and is quite robust over the class of mixed 0–1 programs. We tested it over a standard set of real-world publicly available benchmark instances. In some cases, it not only dominates other general purpose codes, but also algorithms that use the specific structure of the problem. In this paper, we investigate the computational issues that need to be addressed when developing such a procedure. Some of these issues have a theoretical answer but others are of an experimental nature and can only be settled by comparing alternatives on a set of test problems.

Consider the mixed 0–1 program (MIP)

$$\begin{aligned} &\text{Min } cx \\ &\text{subject to} \\ Mx &\geq d \\ x &\geq 0 \\ x_i &\in \{0,1\}, \qquad i = 1,\dots,p \end{aligned}$$

where the first p variables are $0-1$ constrained and the remaining variables x_i, $i = p+1,\dots,n$, are continuous.

At a generic step of the branch-and-cut algorithm, the original linear programming relaxation, $Mx \geq d$, $x \geq 0$, $x_i \leq 1$, $i = 1,\dots,p$, is enriched by additional valid inequalities for (MIP), and some of the 0–1 constrained variables are fixed either at their upper or lower bound. We denote by $\mathcal{C}$ the current family of valid inequalities for (MIP) and we assume that the linear system $Ax \geq b$ defining $\mathcal{C}$ contains at least all the inequalities in $Mx \geq d$, $x \geq 0$, $x_i \leq 1, i = 1,\dots,p$. We denote by $F_0, F_1 \subseteq \{1,\dots,p\}$ the sets of variables that have been fixed at 0 and 1 respectively.

$$\text{Let } K(\mathcal{C},F_0,F_1) = \{x : \begin{aligned} Ax &\geq b \\ x_i &= 0 \quad \text{for } i \in F_0 \\ x_i &= 1 \quad \text{for } i \in F_1\} \end{aligned}$$

and let $LP(\mathcal{C},F_0,F_1)$ denote the linear program

$$\begin{aligned} &\text{Min } cx \\ &x \in K(\mathcal{C},F_0,F_1), \end{aligned}$$

which is assumed to be feasible, with a finite minimum.

The active nodes of the enumeration tree are represented by a list S of ordered pairs (F_0, F_1). Let UB stand for the current upper bound, i.e. the value of the best known solution to (MIP).

Branch-and-cut procedure

(Input c, M, d, p)

1. **Initialization:** *Set $S = \{(\emptyset, \emptyset)\}$, let C consist of the linear programming relaxation of (MIP) and $UB = \infty$.*

2. **Node selection:** *If $S = \emptyset$ stop. Otherwise choose an ordered pair $(F_0, F_1) \in S$ and remove it from S.*

3. **Lower bounding step:** *Solve the linear program $LP(C, F_0, F_1)$. If the problem is infeasible go to Step 2, otherwise let $\bar{x}$ denote its optimal solution. If $c\bar{x} \geq UB$ go to Step 2. If $\bar{x}_j \in \{0, 1\}$, $j = 1, \ldots, p$, let $x^* = \bar{x}, UB = c\bar{x}$ and go to Step 2.*

4. **Branching/cutting decision:** *Should cutting planes be generated? If yes, go to Step 5, else go to Step 6.*

5. **Cut generation:** *Generate cutting planes $\alpha x \geq \beta$ valid for (MIP) but violated by $\bar{x}$. Add the cuts to C and go to Step 3.*

6. **Branching step:** *Pick an index $j \in \{1, \ldots, p\}$ such that $0 < \bar{x}_j < 1$. Generate the subproblems corresponding to $(F_0 \cup \{j\}, F_1)$ and $(F_0, F_1 \cup \{j\})$, calculate their lower bounds and add them to S. Go to Step 2.*

When the algorithm stops, if $UB < \infty$, then x^* is an optimal solution to (MIP), otherwise (MIP) is infeasible. In addition to the steps of the basic branch-and-cut procedure stated above, several steps can be performed periodically to improve efficiency: use of heuristics, preprocessing, variable fixing, etc., but this paper is not concerned with such refinements.

In this paper we investigate the use of general cutting planes in Step 5 of the branch-and-cut procedure. The cuts we use are lift-and-project cuts and some of the major issues that need to be addressed are the following.

- For computational reasons, it is important that the cuts generated at any node of the branch-and-cut tree be valid at all the other nodes.
- Not surprisingly, the degree to which general cutting planes help in solving (MIP) varies a lot depending on the instance. A key issue addressed in this paper is a measure of cut quality. Such a measure is used in this paper for comparing alternative versions of the lift-and-project cuts and, in the branch-and-cut algorithm, for deciding what is the right amount of cutting relative to branching.
- A difficult question is the branching/cutting decision that needs to be made in Step 4 of the algorithm.
- When the decision to generate cuts is made in Step 4, it is often efficient to generate more than one cut. For a given amount of time devoted to generating cuts, is it better to generate cuts sequentially, updating C and $\bar{x}$ before generating a new cut, or is it better to generate the cuts in a batch without updating $\bar{x}$?

A follow up paper will discuss other issues, not dealing directly with lift-and-project cuts, such as the incorporation of Gomory cuts and of some special purpose cuts, general integer variables, heuristics and preprocessing, all of which are part of a code currently under development, called MIPO (this stands for MIP Optimizer).

Section 2 discusses the basic aspects of generating lift-and-project cuts. In particular, it is shown how the cuts generated at any node of the branch-and-cut tree can be made valid at all the other nodes, and how to strengthen the cuts. Section 3 focuses on computational considerations, such as cut quality versus computing time. The issue of whether it is better to generate cuts is small batches or in large batches is also addressed in this section. Section 4 deals with the key decision of branching versus cutting at a node of the enumeration tree. Other decisions such as node and branching variable selection in the enumeration tree are investigated in Section 5. Finally, in Section 6, we compare our optimization procedure with other existing algorithms, such as CPLEXMIP, OSL and MINTO.

2 Cut Generation

In this section we address the problem of generating cutting planes in Step 5 of the branch-and-cut procedure. The use of combinatorial cutting planes, namely, cuts which arise from an underlying combinatorial structure of the problem is well documented in the literature. In contrast, the literature contains no serious attempt to incorporate general cutting planes within branch-and-cut, namely cuts obtained by imposing integrality conditions on one or more of the 0–1 constrained variables. Such general cutting planes include, among others, Gomory cuts [9], intersection cuts [1], disjunctive cuts [2] or lift-and-project cuts [3]. Here we focus on lift-and-project cuts, including several improvements and variations. A major goal of this paper is to compare cuts in terms of some quality measure and of the time necessary to generate them. We start this section by addressing different measures of cut quality. Another key issue addressed in this section is the guarantee that cuts generated at any node of the branch-and-cut tree be valid for all the nodes.

2.1 Measure of Cut Quality

An important problem arising in a computational study of cutting planes is to define meaningful measures for comparing them. It is quite common among integer programmers to compare cutting planes based on the improvement obtained in the objective function once the inequality is added to the formulation. The main drawback of this measure is that it does not take into account improvements in the formulation that do not have an immediate effect on the objective function value. For example, some hard mixed 0-1 programs may not have any integrality gap. Nevertheless, the cutting planes may be very useful in driving the current solution to integrality. In other mixed $0-1$ programs where there is an integrality gap, the difficulty may reside in proving that there is no feasible solution whose objective value equals that of a linear programming relaxation. In this case, the objective function may only start improving after several iterations, even with strong cuts. In general, the behaviour of the cutting planes may vary so much from problem to problem that a meaningful reduction of the integrality gap in one problem may not necessarily be meaningful for another. Geometric measures compare cutting planes $\alpha x \geq \beta$ by establishing a notion of "depth" relative to the solution $\bar{x}$ that they are designed to cut off. For a full dimensional polyhedron, an appealing geometric measure of cut quality is the euclidean distance between $\bar{x}$ and the hyperplane $\alpha x = \beta$, namely the distance between $\bar{x}$ and its orthogonal projection on this hyperplane. It turns out to be a reliable guide (in the use that we make of it later) even when the underlying integer polyhedron is not full dimensional. Thus, it will be our preferred measure when we report computational results in Section 3. Another useful measure in our work is the amount $\beta - \alpha\bar{x}$ by which the cut is violated by $\bar{x}$. Note that this measure assumes that the cut $\alpha x \geq \beta$ has been normalized. We consider mainly two normalizations. For cuts with nonzero right-hand side, we can use

Normalization 1: $|\beta| = 1$.

In general, we can use

Normalization 2: $\sum_i |\alpha_i| = 1$.

2.2 Lift-and-Project Cuts

Let $\bar{x}$ be the solution obtained when solving $LP(\mathcal{C}, F_0, F_1)$, at a generic node (F_0, F_1) of the branch-and-cut tree. In this section, we assume that $\bar{x}$ is not feasible to (MIP), and we let $j \leq p$ be the index of a 0–1 variable such that $0 < \bar{x}_j < 1$. Lift-and-project cutting planes that cut-off $\bar{x}$ are obtained by imposing the 0–1 condition on one or more of the 0–1 constrained variables. Those obtained by imposing the 0–1 condition on variable x_j are valid inequalities for:

$$P_j(K) = \text{conv}(K \cap \{x \in \Re^n : x_j \in \{0,1\}\}),$$

where $K := K(\mathcal{C}, F_0, F_1)$. A theorem of Balas [2] can be used to characterize all valid inequalities for $P_j(K)$, as follows. Let $F = \{1, \ldots, n\} \setminus (F_0 \cup F_1)$ denote the set of free variables at node (F_0, F_1). We will assume, without loss of generality, that $F_1 = \emptyset$, since if $F_1 \neq \emptyset$ then all the variables x_k in F_1 can be replaced by $1 - x_k$ (which amounts to replacing the column A_k and the righthand side b with $-A_k$ and $b - A_k$, respectively). Then $\sum_{i \in F} \alpha_i x_i \geq \beta$ (which we denote by $\alpha^F x^F \geq \beta$) is a valid inequality for $P_j(K)$, if and only if there exist vectors u, v and scalars u_0, v_0 satisfying:

$$\begin{array}{cccccccc}
\alpha^F & -u^F A^F & -u_0 e_j^F & & & \geq & 0 & \\
\alpha^F & & & -v^F A^F & -v_0 e_j^F & \geq & 0 & \\
& u^F b^F & & & & = & \beta & \\
& & & v^F b^F & +v_0 & = & \beta & \\
& & & & u^F, v^F & \geq & 0, &
\end{array} \qquad (1)$$

where e_j^F is the j^{th} unit vector in $\Re^{|F|}$, A^F is the matrix obtained from A by removing the columns A_i for $i \in F_0$ and the rows corresponding to the inequalities $x_i \geq 0$ for $i = 1, \ldots, n$, and $-x_i \geq -1$ for $i \in \{1,\ldots,p\} \cap F_0$, and b^F is obtained from b by removing the components corresponding to the rows removed from A.

More generally, for any $R \subseteq \{1,\ldots,n\}$, let e_j^R be the j^{th} unit vector in $\Re^{|R|}$, let A^R be the matrix obtained from A by removing the columns A_i for $i \notin R$ and the rows corresponding to the inequalities $x_i \geq 0$ for $i = 1,\ldots,n$, and $-x_i \geq -1$ for $i \in \{1,\ldots,p\} \setminus R$, and let b^R be obtained from b by removing the components corresponding to the rows removed from A. Denote by LP(R) the linear program

$$\begin{array}{lllllcl}
\max \beta - \alpha^R \bar{x}^R & & & & & & \\
\text{subject to} & & & & & & \\
\alpha^R & -u^R A^R & & -u_0 e_j^R & & \geq & 0 \\
\alpha^R & & -v^R A^R & & -v_0 e_j^R & \geq & 0 \\
 & u^R b^R & & & & = & \beta \\
 & & v^R b^R & & +v_0 & = & \beta \\
 & & & & u^R, v^R & \geq & 0 \\
 & & & & (\alpha^R, \beta) & \in & S
\end{array} \qquad (2)$$

where S is a normalization set such as Normalizations 1 or 2 defined earlier.

In order to find a valid inequality for $P_j(K)$ that cuts off $\bar{x}$ we proposed in [3] to solve the linear program LP(F). We called its optimal solution (α^F, β), the *deepest cut*. Notice that such a cut is only valid for the current node and its descendants, since the variables x_i, for $i \in F_0$, have been eliminated from the formulation. In the next section we discuss how to make this cut valid for the whole branch-and-cut tree. The reason for introducing the more general linear program LP(R) will become clear then.

Notice that in any optimal solution to LP(R), we have $\alpha_i^R = max\{\alpha_i^1, \alpha_i^2\}$ for all $i \in R$, where

$$\alpha^1 = u^R A^R + u_0 e_j^R$$
$$\alpha^2 = v^R A^R + v_0 e_j^R.$$

With Normalization 1, we must decide whether to set $\beta = 1$ or $\beta = -1$. In either case, α^R can be eliminated (after introducing slack variables in (2)), reducing the size of the linear program to be solved. The drawback of this normalization is that, in some cases, the linear program may be unbounded. Normalization 2 is always applicable and can be implemented by choosing $S := \{(\alpha^R, \beta) : \sum_{i \in R} |\alpha_i^R| \leq 1\}$. The absolute value constraint used here can be linearized by introducing new variables α_i^+, α_i^-, writing $S := \{(\alpha^R, \beta) : \alpha^R = \alpha^+ - \alpha^-;\ \alpha^+, \alpha^- \geq 0;\ \sum_{i \in R}(\alpha_i^+ + \alpha_i^-) \leq 1\}$, and eliminating α^R.

The variables α^R are eliminated from the linear program LP(R) to reduce its size but are easy to reconstruct from the other variables, namely $\alpha_i^R = max\{\alpha_i^1, \alpha_i^2\}$ in case of Normalization 1, $\alpha^R = \alpha^+ - \alpha^-$ in case of Normalization 2.

2.3 Cut Lifting

In general, a cut generated at a node (F_0, F_1) of the enumeration tree is valid only when the variables in $F_0 \cup F_1$ remain at their fixed values. Such a cut may not be valid for (MIP) and therefore it cannot be used in other parts of the tree. From a computational point of view, it is extremely important that the cuts that are generated be made valid throughout the enumeration tree: it not only reduces the need for extensive bookkeeping, but such "shared" cuts may improve the bounds at many nodes of the tree. A cut which is valid at node (F_0, F_1) is made valid for (MIP) by computing appropriate coefficients for the variables x_j such that $j \in F_0 \cup F_1$. This operation is called *lifting* the cut. In this section, we show how to lift lift-and-project cuts (intersection cuts being a special case, they can be lifted as well).

We briefly summarize the results of [3]. Let $\bar{x}$ be the optimum solution obtained when solving $LP(\mathcal{C}, F_0, F_1)$. We will work with a subvector $\bar{x}^R$ of $\bar{x}$ which contains all the fractional components $\bar{x}_j$ for $j \leq p$, all the positive components $\bar{x}_j$ for $j \geq p+1$, and possibly others. We will compute a lift-and-project cut in the space of variables x^R and then lift the inequality into the original space. W.l.o.g. we can assume that if $i \notin R$ then $\bar{x}_i = 0$, since for those i such that $\bar{x}_i = 1$ the variable x_i can be complemented (by replacing A_i and b with $-A_i$ and $b - A_i$, respectively). Let $j \in \{1, \ldots, p\}$ be an index such that $0 < \bar{x}_j < 1$ and consider the inequality $\alpha^R x^R \geq \beta$ generated by the linear program LP(R) corresponding to this definition of R.

The following theorem shows how the inequality $\alpha^R x^R \geq \beta$ can be extended into a valid cut $\alpha x \geq \beta$ in the original space.

Theorem 2.1 *Let (α^R, β) be an optimal solution to LP(R) where $R \supseteq \{i \in \{1, \ldots, p\} : 0 < \bar{x}_i < 1\} \cup \{i \in \{p+1, \ldots, n\} : \bar{x}_i > 0\}$. W.l.o.g. assume $\{i \in \{1, \ldots, p\} : \bar{x}_i = 1\} = \emptyset$. Then* *the inequality $\alpha x \geq \beta$ defined below is valid for (MIP) and cuts off $\bar{x}$.*

$$\alpha_i = \begin{cases} \alpha_i^R & \text{if } i \in R \\ \max\{\alpha_i^1, \alpha_i^2\} & \text{if } i \notin R, \end{cases}$$

where $\alpha_i^1 = u^R A_i^R$, $\alpha_i^2 = v^R A_i^R$ and A_i^R denotes the subvector of A_i with the same row set as A^R.

Proof. Since $i \notin R$ implies $\bar{x}_i = 0$, it is clear that the inequality $\alpha x \geq \beta$ cuts off $\bar{x}$, so we only have to show that $\alpha x \geq \beta$ is valid for (MIP). That is, we must show that $(\alpha, \beta) = (\alpha^N, \beta)$ is a feasible solution of LP(N), where $N = \{1, \ldots, n\}$. The linear programs LP(R) and LP(N) differ in the following ways.

LP(N) contains more variables and constraints, namely the constraints

$$\begin{aligned} \alpha_i^N - u^N A_i^N &\geq 0 \quad \text{for } i \in N \setminus R \\ \alpha_i^N - v^N A_i^N &\geq 0 \quad \text{for } i \in N \setminus R \end{aligned}$$

and the variables u_i and v_i corresponding to the constraints $-x_i \geq -1, i \in N \setminus R$ that had been removed from $A^N x \geq b^N$.

A feasible solution to LP(N) can be obtained from a feasible solution to LP(R) by setting the extra variables u_i and v_i equal to zero, and setting $\alpha_i, i \in N \setminus R$ as stated in the theorem. Then the above extra constraints are satisfied. ■

An important property shown in [3] is that, for Normalization 1, the cut $\alpha^R x \geq \beta$ is "optimally" lifted by Theorem 2.1, in the sense that the resulting cut $\alpha x \geq \beta$ is identical to the one that would be obtained by solving LP(N). The more complicated statement (Theorem 3.2 in [3]) arises because [3] uses an equality version of LP(R). For Normalization 2, however, there is usually a difference between the cuts obtained from LP(R) by lifting and those obtained by solving LP(N).

The lifting procedure of Theorem 2.1 is computationally inexpensive, since each lifted coefficient can be computed in time proportional to the number of rows of A^R. This theorem is crucial to the success of a branch-and-cut algorithm based on lift-and-project cutting planes. First, by choosing R such that $R \cap (F_0 \cup F_1) = \emptyset$, Theorem 2.1 provides an easy and efficient way of making the cuts that are generated at the individual nodes of the branch-and-cut tree valid for the whole tree. Second, it implies that the size of the linear programs that need to be solved for generating cutting planes can be substantially reduced. rendering the approach computationally more efficient. In fact, a good choice for R is to take it as small as possible, that is $R = \{i \in \{1, \dots, p\} : 0 < \bar{x}_i < 1\} \cup \{i \in \{p+1, \dots, n\} : \bar{x}_i > 0\}$.

2.4 Cut Strengthening

The lift-and-project cut $\alpha x \geq \beta$ of Theorem 2.1 is derived from the 0–1 condition on x_j. The cut $\alpha x \geq \beta$ can be strengthened to $\gamma x \geq \beta$ by using the integrality condition on variables other than x_j, as shown by Balas and Jeroslow [4] (see also Section 7 of [2]).

Theorem 2.2 *The inequality $\gamma x \geq \beta$ is valid for (MIP), where*

$$\begin{aligned} \gamma_k &= \min\left\{\alpha_k^1 + u_0\lceil \bar{m}_k \rceil, \alpha_k^2 - v_0\lfloor \bar{m}_k \rfloor\right\} \textit{ for } k = 1, \dots, p, \\ \gamma_k &= \max\left\{\alpha_k^1, \alpha_k^2\right\} \textit{ for } k = p+1, \dots, n, \end{aligned}$$

with α^1 and α^2 as defined in Section 2.2 and

$$\bar{m}_k = \frac{\alpha_k^2 - \alpha_k^1}{u_0 + v_0} \qquad k = 1, \dots, p. \tag{3}$$

Proof. (This version of the proof is due to L. Wolsey) We show that the cut is valid by proving that it arises from a valid disjunction for the 0–1 solutions of MIP. Suppose that in order to derive a cutting plane we use the valid disjunction $-x_j + mx \geq 0 \vee x_j - mx \geq 1$, instead of $-x_j \geq 0 \vee x_j \geq 1$, where m is a vector of all integer components such that $m_k = 0, k = p+1, \dots, n$. Then the coefficients of the cut obtained by imposing the new disjunction are defined as:

$$\begin{aligned} \gamma_k &= \max\left\{\alpha_k^1 + u_0 m_k, \alpha_k^2 - v_0 m_k\right\} \text{ for } k = 1, \dots, p, \\ \gamma_k &= \max\left\{\alpha_k^1, \alpha_k^2\right\} \text{ for } k = p+1, \dots, n. \end{aligned}$$

We can now choose $m_k, k = 1, \dots, p$, as the integers that make $\gamma_k, k = 1, \dots, p$, as small as possible. It is easy to see that these values are obtained by first finding the value of m_k that makes the two terms in the brackets equal, which is $\bar{m}_k$ as in (3), and then taking $m_k = \lfloor \bar{m}_k \rfloor$ or $m_k = \lceil \bar{m}_k \rceil$, whichever gives the minimum value for γ_k. This yields

$$\begin{aligned} \gamma_k &= \min\left\{\alpha_k^1 + u_0\lceil \bar{m}_k \rceil, \alpha_k^2 - v_0\lfloor \bar{m}_k \rfloor\right\} \text{ for } k = 1, \dots, p, \\ \gamma_k &= \max\left\{\alpha_k^1, \alpha_k^2\right\} \text{ for } k = p+1, \dots, n, \end{aligned}$$

and the theorem follows.

■

It is important to note that the cut strengthening is done once the multipliers u, v, u_0, and v_0 are defined by the solution to LP(R), i.e. the disjunction is changed only a posteriori.

3 Computational Testing of the Cuts

This section is devoted to testing several cutting plane strategies. Given a solution $\bar{x}$ of $LP(\mathcal{C}, F_0, F_1)$ which is not feasible to (MIP), we call a *round* of cuts a set of cutting planes generated for some or all $j \in \{1, \ldots, p\}$ such that $0 < \bar{x}_j < 1$. One of the key issues investigated in this section is whether it is better to generate cuts in large or small rounds. Before we deal with this question, we first settle three other computational issues: lifted versus non-lifted cuts, strengthened versus non-strengthened cuts and whether LP(R) is the best linear program we can use to generate the cuts. In each case, the issue boils down to the quality of cuts versus the time required to generate them. Each experiment, except for the last, was conducted as follows: we ran 10 rounds of cuts (or fewer whenever an optimal solution of (MIP) was found) and, in each round, we generated a cutting plane for each fractional component of the current solution. The cut comparisons were performed at the root node, i.e. $F_0 = F_1 = \emptyset$. The cuts generated in a round were sorted according to the amount by which they were violated by $\bar{x}$ (most violated first), and a cut was added to $\mathcal{C}$ if the cosine of the angle between its normal vector and each previously added cut was at most $\theta < 1$, where θ is a parameter. Here we took $\theta = 0.999$, the main purpose being to discard duplicate cuts generated in the same round, if any. The linear programs encountered during the procedure were solved using the CPLEX callable library. The times reported in this paper refer to seconds on an HP720 Apollo desktop workstation with 64 megabytes of memory.

3.1 The Test-Bed

For the purpose of testing computationally both our cuts (in this section) and our branch-and-cut algorithm (in Section 6), we used a wide variety of mixed 0–1 programs arising from applications. Since mixed 0–1 programs can have very different structures, any algorithm

Problem name	Constraints	0–1 variables	Continuous variables	Value of LP optimum	Value of IP optimum
BM23	20	27	0	20.57	34
CTN2	150	120	120	169.79	239.21
EGOUT	98	55	86	149.589	568.101
FXCH3	161	141	141	152.01	197.98
MISC05	300	74	62	2930.9	2984.5
MODGLOB	291	98	324	20430947	20740508
MOD008	6	319	0	290.93	307
P0033	15	33	0	2520.57	3089
P0201	133	201	0	6875.00	7615
P0282	241	282	0	176867.50	258411
P0291	252	291	0	1705.13	5223.749
P2756	755	2756	0	2688.75	3124
SCPC2S	385	468	0	209.85	216
SET1AL	492	240	472	11145.62	15869.7
TSP43	143	1117	0	5611	5620
VPM1	234	168	210	15.4167	20

Table 1: Problem characteristics

for this class of problems must be tested on a wide variety of instances. Many of our test problems are taken from MIPLIB, a publicly available library of real-world mixed integer programs compiled by Bixby, Boyd and Indovina [7]. The problems Pxxxx are pure 0-1 problems. They were first described in Crowder, Johnson and Padberg [8]. SCPC2S is a set-covering problem generated by Beasley [5] and preprocessed by Nobili and Sassano [13]. BM23 is a small but relatively difficult 0-1 problem due to Bouvier and Messoumian. The problems EGOUT, MODGLOB, SET1AL and VPM1 are mixed 0–1 programs with fixed-charge network flow structure, described in Van Roy and Wolsey [15]. CTN2 and FXCH3 are fixed-charge network flow problems described in [3]. MOD008 originated at IBM France and MISC05 comes from circuit design. These problems range from difficult to not-so difficult. We also found interesting to test our algorithm on some pure 0–1 problems arising from combinatorial optimization problems, where special purpose algorithms have been developed and tested. TSP43 is a 43-city asymmetric traveling salesman problem arising from a scheduling problem at DuPont, and was provided to us by D. Miller and J. Pekny. The formulation used here contains the degree constraints and a collection of subtour elimination constraints that, together, provide a tight lower bound (see [3] for a more detailed discussion of this instance). Table 1 contains the characteristics of the problems used in our test bed.

3.2 Cut Lifting

For Normalization 2, there is usually a difference between the cuts obtained from $R = \{1, \ldots, n\}$ and those obtained from $R = \{i \in \{1, \ldots, p\} : 0 < \bar{x}_i < 1\} \cup \{i \in \{p+1, \ldots, n\} : \bar{x}_i > 0\}$ followed by the lifting step of Theorem 2.1. However, the following computational experiment demonstrates that the difference in cut quality between the two choices of R does not justify the enormous increase in time needed to generate the cutting planes in the full space of variables. The results are summarized in Table 2, for $R = \{1, \ldots, n\}$ under the heading "Not lifted" and for $R = \{i \in \{1, \ldots, p\} : 0 < \bar{x}_i < 1\} \cup \{i \in \{p+1, \ldots, n\} : \bar{x}_i > 0\}$ under the heading "Lifted". For both choices of R we report the total number of cuts generated during ten rounds of cutting, the average depth of the cuts (using the geometric measure introduced in Section 2.1, namely the euclidean distance between $\bar{x}$ and the cut hyperplane $\alpha x = \beta$) and the percentage of the integrality gap closed after ten rounds of cuts. First, note that, even though the time spent to generate the "non-lifted" cuts is considerably greater, their quality (using the geometric measure or the gap closed) is sometimes worse.

Problem	Lifted				Not lifted			
	Number of cuts	Average distance	Gap closed	CPU time	Number of cuts	Average distance	Gap closed	CPU time
BM23	103	0.05	39.2%	8	107	0.05	34.8%	23
CTN2	334	0.11	95.8%	240	340	0.12	95.0%	1346
EGOUT	77	0.71	100%	15	84	0.67	100%	39
FXCH3	291	0.14	88.8%	205	304	0.13	89.9%	1999
MISC05	244	0.03	12.4%	284	250	0.02	21.7%	1228
MODGLOB	412	0.31	96.6%	14498	341	0.50	96.4%	56618
MOD008	106	0.01	43.0%	10	88	0.01	32.2%	882
P0033	123	0.05	72.9%	7	126	0.05	69.1%	20
P0201	654	0.03	59.8%	1714	792	0.04	72.2%	24040
P0282	340	0.06	94.1%	125	282	0.07	96.3%	745
P0291	122	0.09	98.8%	8	138	0.06	97.7%	280
P2756	498	0.23	90.3%	163	388	0.3	97.2%	31383
SCPC2S	1132	0.02	60.5%	15171	351	0.03	43.1%	16169 *
SET1AL	257	0.71	99.9%	113	246	0.74	99.9%	961
TSP43	709	0.01	62.2%	2782	505	0.01	63.8%	15000 *
VPM1	300	0.05	72.3%	211	203	0.08	84.5%	655

$*$ space limit or time limit exceeded

Table 2: Cuts, lifted versus not lifted

Second, the time gained by lifting the cuts increases very significantly with the number of variables: for the four largest instances in our test bed (in terms of the number of variables), the non-lifted cuts required CPU time about 90 times greater than lifted cuts in one case, 200 times greater in another case, and had to be terminated in the last two cases due to excessive time or space used. As a consequence of this experiment, our code always solves (2) in the space $R = \{i \in \{1, \ldots, p\} : 0 < \bar{x}_i < 1\} \cup \{i \in \{p+1, \ldots, n\} : \bar{x}_i > 0\}$.

3.3 Cut Strengthening

The following experiment compares the effect of using "strengthened" versus "unstrengthened" cuts. In both cases, the cuts were lifted using Theorem 2.1. The strengthening step, when applied, was performed after lifting. As in the previous experiment we report the number of cuts generated in ten rounds, the average depth of the cuts, the percentage of the integrality gap closed and the CPU time. It can be seen from Table 3 that the strengthening step improves the quality of lift-and-project cuts in most cases, while the time needed to perform it is relatively small. It may seem surprising that, in two or three cases, the effect of strengthening was detrimental in terms of the gap closed or average depth of cuts. But remember that our results are reported after the completion of ten rounds of cuts. After one round, strengthening can only improve the gap closed and the average depth of cuts, but in the following rounds, the fractional points to be cut off are likely to be different. As a consequence, the cuts generated in rounds two to ten may be unrelated and, due to chance factors, the run with unstrengthened cuts may turn out to be the best. This is the exception, however, and the gains that are typically achieved with the strengthening step are well worth the small amount of time required to perform it. Therefore MIPO performs cut strengthening whenever possible.

3.4 Cuts Arising from Weaker Relaxations

The number of variables in the cut generating LP (2) is twice the number of rows of A^R when using Normalization 1 and twice the number of rows plus columns of A^R when using Normalization 2. So, clearly, the effort involved in generating lift-and-project cuts is substantial. In this section we study cuts obtained when a reduced set of constraints defining $K(\mathcal{C}, F_0, F_1)$ is involved in the cut generating LP.

Problem	Strengthened				Not strengthened			
	Number of cuts	Average distance	Gap closed	CPU time	Number of cuts	Average distance	Gap closed	CPU time
BM23	103	0.05	39.2%	8	124	0.03	34.3%	13
CTN2	334	0.11	95.8%	240	331	0.11	95.5%	225
EGOUT	77	0.71	100%	15	87	0.62	100%	19
FXCH3	291	0.14	88.8%	205	284	0.16	82.1%	176
MISC05	244	0.03	12.4%	284	237	0.11	11.7%	243
MODGLOB	412	0.31	96.6%	14498	402	0.29	96.2%	12030
MOD008	106	0.01	43.0%	10	145	0.003	20.9%	24
P0033	123	0.05	72.9%	7	82	0.05	70.3%	2
P0201	654	0.03	59.8%	1714	686	0.02	55.2%	2024
P0282	340	0.06	94.1%	125	305	0.05	95.2%	59
P0291	122	0.09	98.8%	8	128	0.07	95.9%	8
P2756	498	0.23	90.3%	163	472	0.24	97.4%	128
SCPC2S	1132	0.02	60.5%	15171	1199	0.02	59.5%	16559
SET1AL	257	0.71	99.9%	113	257	0.71	99.9%	111
TSP43	709	0.01	62.2%	2782	547	0.01	34.4%	641
VPM1	300	0.05	72.3%	211	290	0.05	62.3%	198

Table 3: Cuts, strengthened versus not strengthened

Problem	Not reoptimized				Semi-reoptimized				Fully reoptimized			
	# of cuts	Avg dist	Gap closed	CPU time	# of cuts	Avg dist	Gap closed	CPU time	# of cuts	Avg dist	Gap closed	CPU time
BM23	58	0.02	27.9%	4	109	0.04	36.4%	5	103	0.05	39.2%	8
CTN2	329	0.08	73.6%	92	325	0.12	93.4%	141	334	0.11	95.8%	240
EGOUT	133	0.41	99.0%	20	93	0.84	100%	12	77	0.71	100%	15
FXCH3	244	0.09	44.3%	40	294	0.12	76.7%	134	291	0.14	88.8%	205
MISC05	170	0.02	7.7%	30	197	0.04	17.2%	58	244	0.03	12.4%	284
MODGL.	443	0.17	67.0%	2254	477	0.28	87.8%	1412	412	0.31	96.6%	14498
MOD008	39	.002	33.5%	6	115	0.01	52.1%	8	106	0.01	43.0%	10
P0033	51	0.04	68.9%	4	136	0.05	74.6%	7	123	0.05	72.9%	7
P0201	513	0.02	24.6%	471	548	0.03	63.5%	408	654	0.03	59.8%	1714
P0282	206	0.05	23.4%	15	346	0.05	94.7%	71	340	0.06	94.1%	125
P0291	97	0.06	90.5%	7	122	0.08	98.5%	5	122	0.09	98.8%	8
P2756	184	0.20	3.3%	42	448	0.26	97.6%	80	498	0.23	90.3%	163
SCPC2S	556	0.01	12.2%	534	1465	0.01	47.5%	10551	1132	0.02	60.5%	15171
SET1AL	278	0.66	99.9%	68	261	0.70	99.9%	99	257	0.71	99.9%	113
TSP43	423	0.01	55.6%	308	380	0.02	55.6%	257	709	0.01	62.2%	2782
VPM1	345	0.02	23.6%	89	301	0.05	72.7%	95	300	0.05	72.3%	211

Table 4: Comparison of three types of cuts

Table 4:

The first relaxation of $K(C, F_0, F_1)$ that we consider only contains constraints which are tight at the current solution $\bar{x}$. Specifically, we only keep the constraints whose slacks are nonbasic in the solution of $LP(C, F_0, F_1)$. The cuts obtained from this relaxation are exactly the intersection cuts associated with the convex set $0 \leq x_j \leq 1$. In Table 4, these cuts are reported under the heading "Not reoptimized". It was shown in [3] that these cuts can be generated directly from the basis inverse of $LP(C, F_0, F_1)$. This is extremely fast. In our experiment, we did not do this but solved linear program (2) instead, so the times reported in the column "Not reoptimized" can be greatly improved. The second relaxation reported in Table 4 contains the constraints which are tight at $\bar{x}$ together with the inequalities $x_i \geq 0, i \in R$, $x_i \leq 1$ for $i \in \{1, \ldots, p\} \cap R$. We call these cuts "Semi-reoptimized". Finally the cuts generated from the full description of $K(C, F_0, F_1)$, as discussed in Theorem 2.1, are called "Fully reoptimized". As before, we report the results obtained after ten rounds of cuts. Remember that, in every round, a cut is generated for each of the basic variables $x_j, j = 1, \ldots, p$ such that $0 < \bar{x}_j < 1$ in the current solution. Building on the conclusions from earlier sections, in this set of experiments the cuts are generated in the subspace $R = \{i \in \{1, \ldots, p\} : 0 < \bar{x}_i < 1\} \cup \{i \in \{p+1, \ldots, n\} : \bar{x}_i > 0\}$ and are strengthened.

The following observations can be made from Table 4. The intersection cuts (under the heading "Not reoptimized") are substantially worse than the semi-reoptimized or fully reoptimized lift-and-project cuts. Quite surprisingly, the semi-reoptimized cuts are not always dominated by the fully reoptimized cuts (both in terms of gap closed and average distance cut off). In fact, quite to the contrary, in about half the cases, the semi-reoptimized cuts are stronger than the fully reoptimized cuts. We offer the following explanation for this unexpected behavior. Even though a fully reoptimized cut is typically deeper than a semi-reoptimized cut, it also tends to be "more parallel" to the objective function. So, as a family, fully reoptimized cuts may not improve the polyhedron as much as semi-reoptimized cuts. In subsequent rounds, having cuts that improve the polyhedron in diverse directions is important for the generation of new cuts. So, as a family, the semi-reoptimized cuts are often more effective. To illustrate this point, consider problem P0201. For each cut that we generated, we computed the cosine of the angle formed by its normal direction and the objective direction, and we averaged these cosines over all cuts generated in a round. In the first round, the average cosine was 0.439 for the semi-reoptimized cuts and 0.565 for the fully reoptimized cuts. So indeed, for this instance, the fully reoptimized cuts were "more parallel" to the objective direction.

As a consequence of this experiment, we use the semi-reoptimized lift-and-project cuts in our code.

3.5 Number of Cuts at a Given Node

The linear programs (2) used to generate cuts for different indices $j \in \{1, \ldots, p\}$ such that $0 < \bar{x}_j < 1$ differ very little, given $K(\mathcal{C}, F_0, F_1)$ and $\bar{x}$. So, once a cut has been generated for some j_1 where $\bar{x}_{j_1}$ is fractional, it may be efficient to generate cuts for other fractional components $\bar{x}_{j_k}$ using the solution obtained from $\bar{x}_{j_1}$ as a "hot start" for the next linear program. This is particularly true if the LP's are solved using the primal simplex method. In our implementation, we actually solve the cut generating LP's using the dual simplex method because degeneracy appears to be less frequent, so the advantage of a "hot start" is reduced. An issue for experimental testing is whether the cuts generated from the various $j \in \{1, \ldots, p\}$ such that $0 < \bar{x}_j < 1$ are sufficiently different from each other to warrant generating them in rounds. In other words, we are again confronted with the issue of quality versus time, but at a more aggregate level than in the previous section.

The time needed for generating a given number of cuts is usually greater with many rounds of few cuts than with fewer rounds of more cuts, because of the need for reoptimization after the generation of every round. Hence we designed our computational experiment as follows: First we ran 5 rounds of cuts, where in every round we generated a cut for every fractional component $\bar{x}_j$, $j = 1, \ldots, p$, of the current LP relaxation solution. Let TTIME denote the total time taken to generate the cuts in these 5 rounds. We then ran for TTIME an experiment where in every round we generated cuts for 50% of the fractional components and for 10%. The fractional components closest to 1/2 are those selected to generate the cuts. In Table 5 we report for each alternative the number of cuts generated, the percentage of the gap closed, and the average distance cut off by the cuts. Although none of the three approaches is uniformly dominant, the first approach closes the largest gap in 9 out of the 16 instances. In some cases, the difference between the three runs is enormous. This is the case for P2756 (see Table 5). It is useful to study this example in greater detail and to look at the evolution of the gap closed over time for each of the three runs. In Figure 1, the objective function value is plotted as a function of time. For several rounds, the cuts do not improve much the objective function value, but they eventually become very effective. The point in time where the objective value starts improving significantly occurs sooner

Problem	100%			50%			10%		
	# of cuts	Avg dist	Gap closed	# of cuts	Avg dist	Gap closed	# of cuts	Avg dist	Gap closed
BM23	44	0.07	31.8%	41	0.06	32.8%	33	0.05	24.6%
CTN2	138	0.22	84.4%	120	0.20	77.3%	70	0.16	46.8%
EGOUT	93	0.61	99.7%	77	0.58	73.2%	44	0.47	36.8%
FXCH3	117	0.19	63.0%	91	0.21	48.9%	67	0.14	31.8%
MISC05	85	0.04	4.3%	62	0.03	3.7%	48	0.03	5.7%
MODGL.	223	0.44	79.8%	227	0.30	73.5%	271	0.11	24.7%
MOD008	38	0.01	34.4%	29	0.01	22.8%	19	0.01	11.2%
P0033	54	0.09	67.9%	60	0.10	47.2%	52	0.09	25.8%
P0201	181	0.06	32.2%	184	0.05	55.4%	85	0.03	44.7%
P0282	157	0.09	88.9%	158	0.08	94.1%	85	0.10	92.3%
P0291	57	0.16	89.0%	49	0.13	98.0%	27	0.14	69.1%
P2756	326	0.35	62.1%	295	0.26	4.6%	122	0.03	0.3%
SCPC2S	663	0.02	37.6%	470	0.02	38.2%	309	0.02	40.0%
SET1AL	233	0.78	99.9%	207	0.76	91.5%	111	0.73	50.9%
TSP43	95	0.03	19.4%	46	0.03	16.1%	26	0.02	26.7%
VPM1	126	0.07	34.8%	119	0.07	30.2%	67	0.06	13.8%

Table 5: Large versus small rounds of cuts

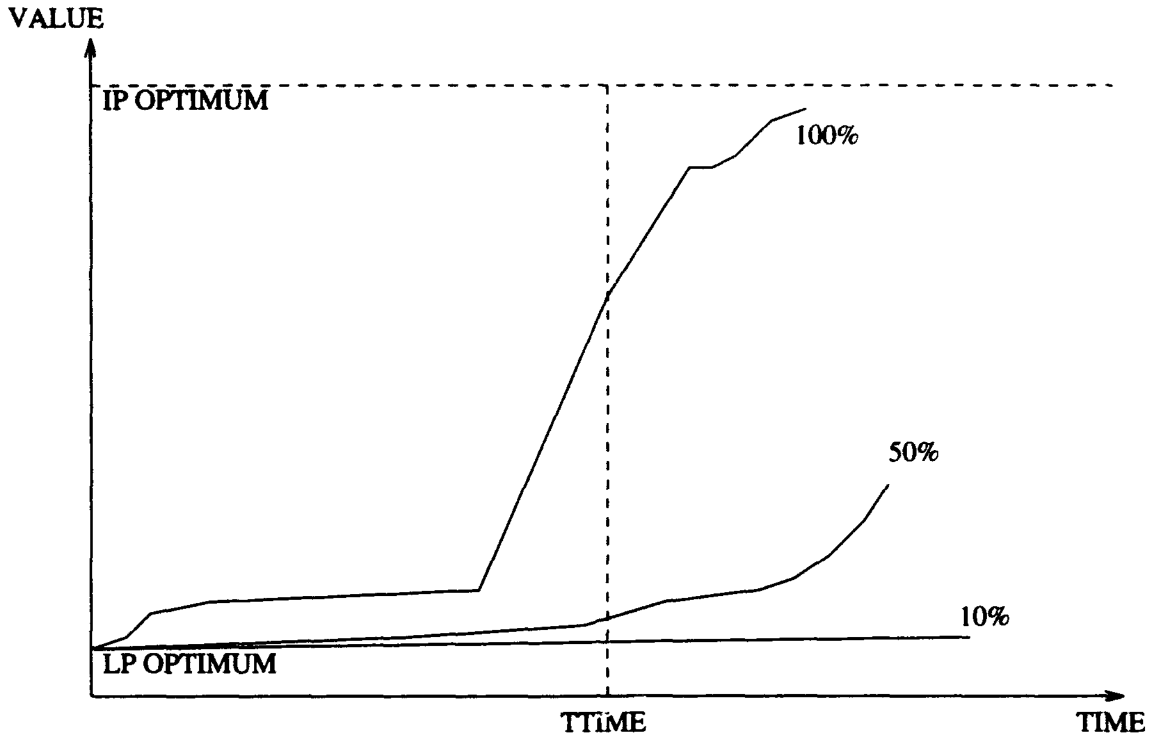

Figure 1: P2756: objective function value as a function of time

when the cuts are generated in large rounds. Although P2756 is an extreme case, several other instances exhibit a similar, less pronounced, behaviour. For other problems, such as SCPC2S, generating cuts for 100% of the fractional variables is inferior after five rounds (the choice we made for TTIME in Table 5), but becomes superior for a larger number of rounds.

Based on this set of experiments, our current implementation generates rounds of semi-reoptimized cuts for all the 0–1 variables which are fractional in the current solution $\bar{x}$ or some upper bound MAXROUND, whichever is smaller. In our implementation, we use MAXROUND=40 for instances with up to 400 0–1 variables and MAXROUND=80 for problems with more 0–1 variables.

3.6 Pool Management

A successful implementation of branch-and-cut requires paying attention to the management of the cuts that have been generated in the course of the algorithm. If all the generated cuts were kept active in the problem formulation the linear programs $LP(\mathcal{C}, F_0, F_1)$ would become too large and sometimes difficult to solve. This is the main reason why the generated cuts are kept in two separate lists, the *active list* and the *pool*. In the active list we keep the constraints which are currently active in the formulation, while the pool contains all other generated cuts. In the initial step both lists are empty. After a first set of cuts is generated, they are all added to the list of active cuts, and periodically, cuts that become inactive are transfered to the pool and deleted from the active list. On the other hand, once a node (F_0, F_1) is retrieved from the queue, the pool is searched for cuts violated by the solution $\bar{x}$ retrieved at this node. These violated inequalities are added to the active list and removed from the pool.

We usually put an upper limit on the size of the pool. If the number of constraints in the pool becomes larger than the upper limit some of the inequalities are deleted. A typical value used for this upper limit is 500 constraints.

3.7 A computational issue

A computational issue related to the solution of the linear programs LP(R) is worth mentioning here.

One of the most common problems found during the solution of these linear programs was the presence of unbounded solutions, even when Normalization 2 was used. This is to be expected when Normalization 1 is used, as discussed in [3], but should not be the case whenever Normalization 2 is used. We could find the following possible explanation for this phenomenon. The linear programs are presumably not really unbounded, but the LP solver finds them unbounded because of numerical difficulties. These may originate either in degeneracy, or in the fact that the reduction step eliminates variables close to, but not actually at their upper or lower bounds. For example, it is typically the case that variables which are at values close to zero, but not exactly zero (say 10^{-5}) are not included in the set R.

The appearance of an unbounded solution is handled in practice by taking as the optimal solution to LP(R) the last basic feasible solution given by the LP solver. Our experience is that the cuts obtained in this manner are still quite deep, probably because in most cases the last basic feasible solution is indeed the optimal one. If the cut obtained in this fashion is not violated by the current solution, it is discarded.

4 Cutting/Branching Decision

One of the key decisions to be taken in branch-and-cut procedures is in Step 4: the branching versus cutting decision. In branch-and-cut algorithms that use combinatorial cutting planes, this decision is relatively easy due to two factors: first, restricted classes of inequalities are considered as cutting planes and, second, the heuristics used to find violated inequalities from these restricted classes might only manage to generate a small fraction of them, if any. With cutting plane procedures guaranteed to find a cut, the branching versus cutting decision has to be confronted. Indeed, pure cutting is possible, but often at a prohibitive computational cost. At the other extreme, pure branch-and-bound may also fail, whereas an appropriate balance between branching and cutting may result in small computing times.

We have shown in the previous section that when we decide to generate a cut by lift-and-project, it is usually computationally efficient to generate a full round of cuts. So, we generate cuts in rounds of cardinality equal to the number of fractional $0-1$ variables, or the upper bound MAXROUND, whichever is smaller. In this section, we address the following questions: Should cuts be generated at all the nodes of the enumeration tree? If cuts are not generated at all the nodes, how frequently should they be generated?

We call a *fixed strategy* one where a round of cuts is generated every k nodes of the enumeration tree, for some fixed integer k (called the *skip factor*). These fixed strategies are used as a benchmark for comparison. We then consider a problem dependent *automatic strategy* in the sense that the skip factor is chosen automatically by the algorithm, depending on the instance to be solved. More sophisticated strategies would be *adaptive strategies* where the skip factor may vary throughout the enumeration tree: when the cuts in C are considered effective according to the measure, rounds of cuts are generated more frequently and when the effectiveness measure is small, cuts are generated with less frequency. We first investigate fixed strategies.

Table 6 compares computing times for our test problems with various cut frequencies. We report the CPU times for skip factors $k = 1, 2, 4, 8, 16, 32, 64$, 128 and when no cuts at all are generated, i.e. our code is run as a pure branch-and-bound code. The last column gives the average depth of the cuts generated at the root node. In order to reduce the variations in computing time due to factors other than k, we used "best bound" as the node selection strategy, and we set the initial upper bound UB equal to the optimum value of (MIP). An immediate observation from Table 6, comparing columns "k=128" and "No cut", is that

Problem	$k = 1$	$k = 2$	$k = 4$	$k = 8$	$k = 16$	$k = 32$	$k = 64$	$k = 128$	No cut	Avg dist
BM23	16.8	9.5	4.9	3.6	2.8	2.8	2.7	2.5	2.3	0.11
CTN2	220	223	153	118	132	95	183	179	***	0.38
EGOUT	9.1	8.3	8.6	8.9	11.2	15.0	23.9	37	4927	0.92
FXCH3	287	195	137	101	109	120	169	216	***	0.36
MISC05	740	270	170	101	112	94	93	95	73	0.06
MODGL.	3584	***	***	2370	***	1703	1657	3164	***	0.85
MOD008	244	191	171	142	96	111	155	211	407	0.02
P0033	7.2	3.1	2.0	1.3	2.7	2.1	2.8	3.9	3.2	0.29
P0201	1674	926	432	177	163	112	115	96	85	0.07
P0282	72.6	65.7	39.8	24.8	40.3	59.0	107	351	***	0.20
P0291	4.9	4.5	9.2	8.0	17.4	47.0	53.8	128	***	0.24
P2756	944	1894	2211	12618	***	***	***	***	***	0.38
SCPC2S	1296	705	239	247	91	76	67	66	38	0.03
SET1AL	134	125	131	165	222	294	542	899	***	0.81
TSP43	954	429	238	196	164	533	354	598	208	0.03
VPM1	4621	8010	15078	3523	2502	3320	4763	4096	***	0.07

$***$ space limit or time limit exceeded

Table 6: Computing times for various skip factors k

many problems that could not be solved by pure branch-and-bound became manageable when cuts were added at the root node and then sporadically throughout the enumeration tree. More importantly, Table 6 indicates a correlation between the fixed strategy that yields the best time and the average depth of the cuts generated at the root node: the deeper the cuts at the root node, the more frequently we should cut in the branch-and-cut algorithm. To illustrate this point, Figure 2 plots the computing time as a function of the skip factor, for nine of the sixteen instances in Table 6. Each curve represents a problem and all computing times have been normalized so that, on each curve, the smallest time equals one. The curves are labeled by the average distance d cut off by the cuts at the root node (refer to Table 6). Figure 2 shows that, by using a fixed strategy of $k = 8$ or $k = 16$, eight of the nine instances are solved within a factor of two of the best computing time over all fixed strategies. Furthermore, a pattern clearly emerges from Figure 2, with the instances having largest d solved fastest using the smallest skipping factors. Unfortunately, the correlation is far from being perfect: the reader can use the data from Table 6 to plot the seven missing curves. This should not surprise anyone familiar with the diversity of instances of (MIP) and the difficulty of devising a robust MIP solver. Nevertheless, there is a clear (negative) correlation between the best skip factor k and the measure d of cut quality.

When the cuts are very effective, it might be worth while not only to use a skip factor of $k = 1$ but even to generate several rounds of cuts in each node of the enumeration tree. Indeed, for P2756, we obtain substantially better results this way. However, in our experience, this opportunity rarely presents itself and therefore, in our current implementation, we never generate more than one round of cuts per node.

The times in Table 6 were obtained using semi-reoptimized cuts. The argument used in Section 3.4 in favor of the semi-reoptimized cuts was based mainly on cut quality. Here we reconsider this decision based on time: the intersection cuts are extremely fast to generate using the basis inverse, so we reran the experiments of Table 6 using intersection cuts instead. Even with almost no computing time spent generating the cuts, the overall computing time was greater for ten of the sixteen instances. In fact, four of the instances could not even be solved within the time limit. So we conclude that the additional time spent generating the semi-reoptimized cuts is well worth it.

Next we turn to the design of an automatic strategy. We devised our automatic strategy by relating the value of the best skip factor from Table 6 to our measure of cut quality and some other problem dependent parameters. First, the average distance cut off by the cuts

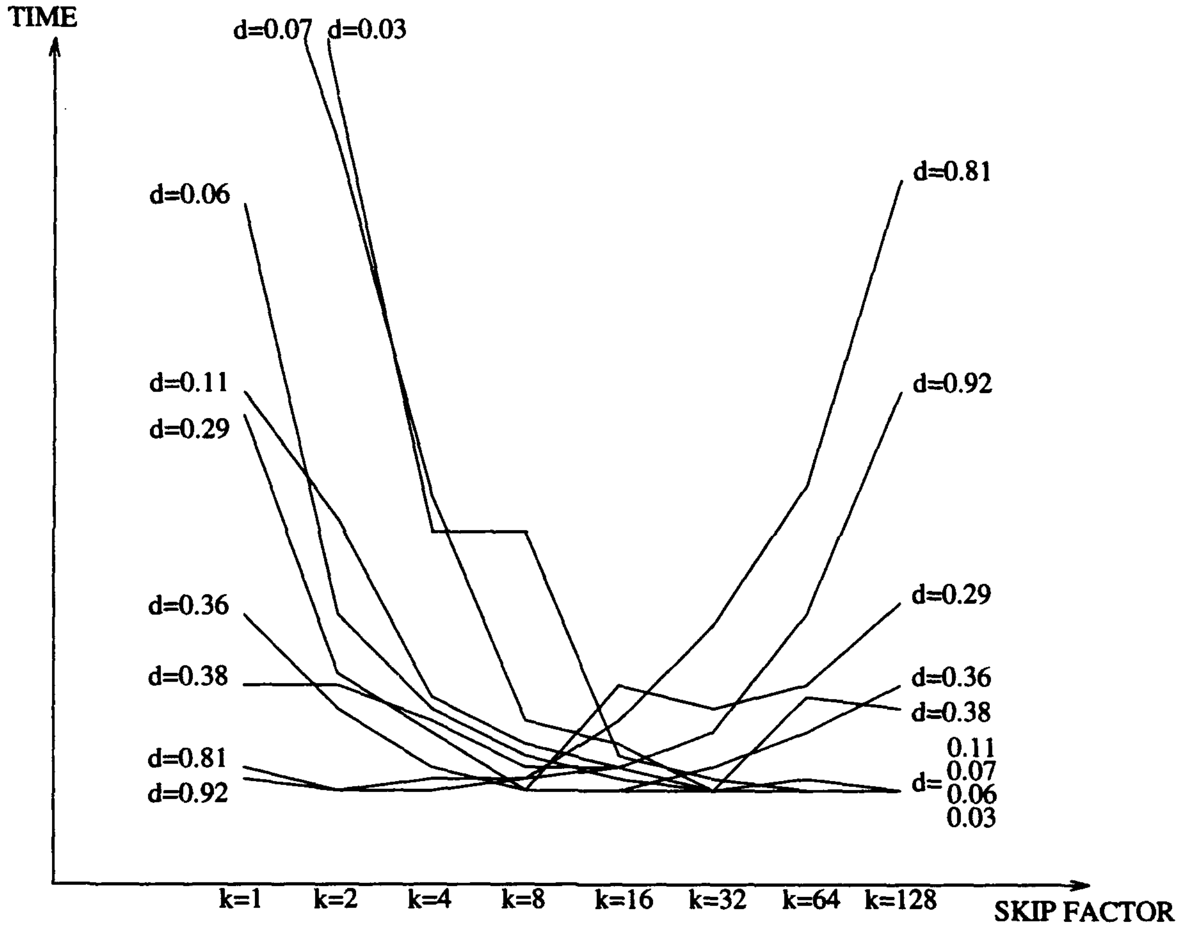

Figure 2: Normalized computing time as a function of the skip factor

at the root node, which we denote by d, is a key parameter used in setting the skip factor k for the automatic strategy. We have observed that d is a reasonably good indicator of the average distance cut off by the cuts generated overall throughout the branch-and-cut tree (d is typically two to three times greater, as the cuts have a general tendency to become shallower when the process advances). The next observation is that, in the best run from Table 6, the ratio $\frac{\textit{number of cuts}}{\textit{number of nodes}}$ for the full branch-and-cut tree is a function not only of the skip factor k but also of the average number of cuts generated in a round, and it is this ratio that we feel should be proportional to d. Since k needs to be set at the root node, we use the number of cuts generated at the root node as a proxi for the average size of a round. Denote this parameter by f. So f is the minimum of MAXROUND and the number of fractional variables at the root node. Finally, it appears that problem size also affects the optimal skip factor: given d and f, the skip factor should be made slightly smaller as the number of 0–1 variables increases. Based on these observations, we chose

$$k = min\ \{\text{KMAX}, \lceil \frac{f}{cd \log_{10} p} \rceil\}$$

where KMAX and c are constants. In our current implementation we set KMAX=32 and $c = 5$. The reason for never using a skip factor greater than KMAX=32 is robustness. When using a skip factor equal to 32, the fraction of time spent generating cuts is small but, in some cases, these cuts make the difference between being able to solve the problem or not, even when the cut quality as measured by d is poor.

As to adaptive strategies, we experimented with different choices, but we could not find a strategy that obtains overall better results than the automatic strategy. Some simple improvements on the automatic strategy should work, such as recomputing the skip factor k after a new round of cuts has been generated with parameters d and f redefined dynamically. We tried more ambitious strategies that favor generating cuts at the higher nodes of the tree (those closest to the root), but these efforts were disappointing. For example, we tried an adaptive strategy that generates a round of cuts at a node with a probability which is inversely proportional to d and the level of the node in the enumeration tree.

5 Enumeration Strategies

5.1 Node Selection

In all the experiments we used the "best bound" rule for node selection. This is done by listing the nodes in the queue in increasing order of objective function value and always picking the first one. It is important to note, however, that these values were computed when the node was generated and they do not reflect improvements obtainable from more recently generated cutting planes. This implies that in general the ordering of the queue is just an approximation to the best bound criterion.

5.2 Branching Variable Selection

If the current node cannot be discarded, a variable is selected for branching. In order to select the branching variable we use the following simple criterion proposed by Padberg and Rinaldi in [14]: we first determine the largest value of $\bar{x}_j \leq 0.5$ and the smallest value $\bar{x}_j \geq 0.5$. Let f_0 and f_1 be the respective values (if f_0 or f_1 is undefined, set it equal to 0.5). We define the set of candidate variables for branching as

$$I = \{i \in \{1, \ldots, p\} : \frac{f_0}{2} \leq \bar{x}_i \leq \frac{1+f_1}{2}\}.$$

Then we select a variable in I with the largest (in absolute value) objective function coefficient.

6 Computational Experience

We compare our code with CPLEX's branch-and-bound code (CPLEXMIP 2.1), with MINTO 1.4 (using CPLEX 2.1 as the LP solver) and with OSL (subroutine EKKMPRE).

6.1 Expanded Test-Bed

In addition to the test problems given in Table 1, we used some difficult problems from the literature to test our branch-and-cut code. The problem characteristics of these additional

Problem name	Constraints	0–1 variables	Continuous variables	LP value (preprocessed)	Value of IP optimum
AIR04	823	8904	0	55535.436	56137
AIR05	426	7195	0	25877.609	26374
CFAT200-1	1919	200	0	-14.517	-12
CTN3	182	142	142	388.31	432.28
GENOVA6	98	904	0	10213.88	10267
L152LAV	97	1989	0	4656.36	4722
LSEU	28	89	0	947.96	1120
MISC07	212	259	1	1415	2810
MOD010	146	2655	0	6532.08	6548
P0548	176	548	0	315.29	8691
RGN	24	100	80	48.8	82.2
SAN200-0.9-3	1126	200	0	-49.9676	-44
SET1AL	492	240	472	11651.63	15869.75

Table 7: Problem characteristics

instances are given in Table 7. They were obtained from the following sources: the CFAT and SAN instances are maximum stable set problems from the 1993 DIMACS computational challenge (the CFAT problem is a fault diagnosis problem originating with Berman and Pelc [6] and the SAN problem is from Laura Sanchis). The GENOVA instance is a set covering problem obtained from Antonio Sassano. AIR04 and AIR05 are set partitioning problems originating from crew scheduling at American Airlines. L152LAV, LSEU and MOD010 originated at IBM Yorktown Hights. RGN and SET1AL are fixed charge problems due to Laurence Wolsey. The three remaining instances (CTN, MISC and P0548) are from classes already discussed in connection with Table 1. Most of these instances were obtained from MIPLIB. Note, however, that not all the instances in MIPLIB are part of our test bed. Instances were excluded for one of two reasons. Either they are "easy" for branch-and-bound, or several representatives from that problem class are already in the test bed.

Since preprocessing may play a big role in some cases, we report all the results with preprocessed problems (a case in point is P0548: this is why we excluded it from the experiments of Sections 3 and 4). All instances are preprocessed using the MINTO preprocessor. These preprocessed instances are fed to the four codes compared in Tables 8 and 9. The reason for this choice is to focus the comparison on the algorithms themselves. All computational experiments were first run with Normalization 1. In case the cuts generated were not violated by the current solution, the same problem was re-run with Normalization 2. In the end Normalization 1 was used successfully in twenty of the twenty eight instances.

6.2 Comparison between MIP solvers

In this section, we compare a preliminary version of our code, called MIPO, with several existing codes. OSL was run on a RISC 530, whereas the three other codes were run on an Apollo HP720. All times are reported in seconds. The conversion factor between a RISC 530 and an Apollo HP720 is roughly one to one.

Our purpose here is to show that MIPO is robust. Only the semi-reoptimized lift-and-project cuts are used throughout the runs of MIPO. For most instances, not having a good estimate of UB is not a big handicap. However, for some of the very large problems, such as the AIRxx problems, it does make a difference. For this reason, we implemented rudimentary heuristics: at each node where cuts are generated, MIPO uses a "diving heuristic" in the spirit of the one described in Hoffman and Padberg [11]. The subproblem is solved and all

Problem	OSL		CPLEXMIP2.1		MINTO			MIPO		
	# of nodes	CPU time	# of nodes	CPU time	# of nodes	# of cuts	CPU time	# of nodes	# of cuts	CPU time
AIR04	***	***	***	***	***	***	***	308	394	7662
AIR05	***	***	***	***	***	***	***	886	1123	18734
BM23	35	11	571	2	922	155	17	254	72	4
CFAT200-1	4	2726	30	962	12749	0	44791	76	80	848
CTN2	***	***	41071	1300	165	197	23	188	305	86
CTN3	***	***	***	***	302	4007	650	496	625	243
EGOUT	1	1.7	84	0.6	19	3	0.6	16	14	1.3
FXCH3	***	***	***	***	241	565	86	880	587	237
GENOVA6	3787	2375	18826	2006	7533	0	1281	3970	2111	1508
L152LAV	1842	7413	27255	5591	***	***	***	4716	3457	4772
LSEU	87	641	24995	674	153	161	13	1040	112	24
MISC05	454	220	948	37	100	505	79	994	4	119
MISC07	***	***	40713	3832	479	537	297	7766	2354	2634
MOD008	53	107	15203	263	537	479	293	1376	215	50
MOD010	18	212	577	146	6	47	970	18	30	58
MODGLOB	***	***	***	***	360	263	56	960	3346	7413
P0033	8	2.6	1058	2.5	34	15	0.8	138	23	1.6
P0201	164	282	2476	63	116	1233	160	998	292	145
P0282	210	341	***	***	438	3367	1007	218	186	36
P0291	8	10	***	***	101	31	8.5	60	185	17
P0548	0	12	***	***	***	***	***	6422	2433	11036
P2756	37	305	***	***	875	1188	1867	724	595	1933

* * * space limit or time limit exceeded

Table 8: Comparison of OSL, CPLEXMIP, MINTO and MIPO

Problem	OSL		CPLEXMIP2.1		MINTO			MIPO		
	# of nodes	CPU time	# of nodes	CPU time	# of nodes	# of cuts	CPU time	# of nodes	# of cuts	CPU time
RGN	2836	284	5213	49	215	3607	255	846	134	55
SAN200-0.9-3	***	***	2149	4074	***	***	***	310	201	995
SCPC2S	202	277	673	123	833	0	600	62	137	93
SET1AL	***	***	***	***	400	27	10	68	201	161
TSP43	***	***	***	***	***	***	***	746	568	324
VPM1	***	***	***	***	302	4005	650	2934	1438	1848

* * * space limit or time limit exceeded

Table 9: Comparison of OSL, CPLEXMIP, MINTO and MIPO (continued)

OSL		CPLEXMIP2.1		MINTO		MIPO	
first	second	first	second	first	second	first	second
12	4	0	2	10	4	6	16

Table 10: Ranking by number of seach tree nodes

variables at 0 or 1 are fixed at these values. The other 0, 1 variables are then fixed using the branching rule described in Section 5.2, and only the most promising branch is followed. This fixing step is repeated until either an infeasible problem occurs or an integer solution is found. Also, at each node of the enumeration tree, we use two rounding heuristics. In one we round to the closest integer and in the other we round up. By incorporating these heuristics, the computing times deteriorate somewhat for several instances, but overall the resulting code is more robust.

Next we summarize some of the major observations that can be made from Tables 8 and 9.

In terms of the number of nodes in the enumeration tree, the performance of the four codes is aggregated in Table 10 where, for each code, we give the number of times it came

OSL		CPLEXMIP2.1		MINTO		MIPO	
first	second	first	second	first	second	first	second
2	3	5	5	11	3	11	13

Table 11: Ranking by CPU time

in first (smallest number of nodes) and the number of times it came in second. As seen from the table, OSL achieves the smallest number of nodes in twelve of the twenty eight instances. The literature on branch-and-cut often stresses the need to reduce the number of enumerated nodes as much as possible. This seems to be the strategy followed by OSL. Our attitude is that more attention needs to be paid to the trade-off between node enumeration and cut generation. There are situations where enumerating nodes is more efficient than generating cuts. Unfortunately, OSL does not report the number of cuts it generates.

As far as computing time is concerned, Table 11 summarizes the performance of the four codes. While both MINTO and MIPO achieved shortest computing times on the same number of instances (eleven out of twenty eight), MIPO was in second place thirteen times against three times for MINTO.

Overall, the most robust code was MIPO, which solved all twenty eight instances and was best or second best in computing time in twenty four of the twenty eight instances. Of course, there were instances that MIPO could not solve, such as P6000 (a problem from Hoffman and Padberg [10]), but the other codes failed as well, so we did not include such instances in the tables since there is nothing to report for any of the codes.

MINTO did particularly well on the fixed charge problems (CTN2, EGOUT, FXCH3, MOD008, SET1AL, VPM1). Indeed, MINTO generates special purpose cuts for this class of problems.

MIPO performed particularly well on the very large instances, that is those with more than a thousand variables or constraints (AIR04, AIR05, CFAT200-1, L152LAV, MOD010, SAN200-0.9-3, TSP43). Surprisingly, MIPO even managed to outperform MINTO on some instances, like CTN3, where MINTO exploits the problem-specific structure.

In some cases, our general purpose code was competitive with special purpose codes. For example, Hoffman and Padberg [11] give the following results for AIR04 and AIR05 on a

RISC 550: AIR04 required 14441 seconds, 90 nodes and 3787 cuts whereas AIR05 required 139337 seconds, 494 nodes and 15449 cuts. In both cases we were able to solve these instances with a few more nodes and a lot fewer cuts. Another example is TSP43. It can be converted into a symmetric TSP and solved using a special purpose code. The resulting computing time is of a similar order of magnitude as with MIPO [12].

7 Conclusions

In this paper, we investigated the use of lift-and-project cutting planes in a branch-and-cut procedure. The major conclusions are:

- A useful measure of cut quality can be obtained by simply computing the euclidean distance between the hyperplane defining the cut and the point it cuts off.
- It is better to generate cuts in large rounds rather than one at a time or in small rounds.
- Contrary to a wide-spread belief, it is efficient to incorporate general cuts within branch-and-cut.
- The branching/cutting decision is one of the most important issues in the implementation of an efficient general purpose branch-and-cut solver. Small enumeration trees do not always correspond to smaller computing times: it is more important to have the right balance between cutting and branching. This decision seems difficult to automate satisfactorily for all instances. We found that a simple strategy, based on the quality of the cuts at the root node, performs well in most instances.
- Our branch-and-cut code is an efficient solver for mixed 0–1 programs. On several instances of pure and mixed 0–1 programs, it performs as well as, or better than the best currently available mixed integer programming codes. The main advantage of our code is that it is able to generate cutting planes independently of the structure of the problem, thus yielding a more robust MIP solver. In some cases it is even more efficient than state-of-the-art algorithms that make use of the problem-specific structure.

References

[1] E. Balas, Intersection cuts – A new type of cutting planes for integer programming, *Operations Research 19* (1971) 19-39.

[2] E. Balas, Disjunctive programming, *Annals of Discrete Mathematics 5* (1979) 3-51.

[3] E. Balas, S. Ceria and G. Cornuéjols, A lift-and-project cutting plane algorithm for mixed 0-1 programs, *Mathematical Programming 58* (1993) 295-324.

[4] E. Balas and R. Jeroslow, Strengthening cuts for mixed integer programs, *European Journal of Operations Research 4* (1980) 224-234.

[5] J.E. Beasley, OR-library: distributing test problems by electronic mail, Technical Report, Imperial College, The Management School, to appear in *Journal of the Operational Research Society.*

[6] Berman and Pelc, Distributed Fault Diagnosis for Multiprocessor Systems, *Proceedings of the 20th Annual Symposium on Fault-Tolerant Computing*, 340-346.

[7] R.E. Bixby, E.A. Boyd and R.R. Indovina, MIPLIB: A test set of mixed integer programming problems, *SIAM News* (March 1992) 16.

[8] H. Crowder, E. Johnson, M. Padberg, Solving large-scale zero-one linear programming problems, *Operations Research 31* (1983) 803-834.

[9] R. Gomory, An algorithm for the mixed integer problem, technical report RM-2597, The Rand Corporation (1960).

[10] K.L. Hoffman and M. Padberg, Techniques for improving the LP-representation of zero-one linear programming problems, *ORSA Journal on Computing 3* (1991) 121-134.

[11] K.L. Hoffman and M. Padberg, Solving airline crew scheduling problems by branch-and-cut, *Management Science 39* (1993) 657-682.

[12] M. Jünger, private communication.

[13] P. Nobili and A. Sassano, private communication.

[14] M. Padberg and G. Rinaldi, A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems, *SIAM Review 33* (1991) 60-100.

[15] T.J. Van Roy and L.A. Wolsey, Solving mixed integer programming problems using automatic reformulation, *Operations Research 35* (1987) 45-57.